THE OFFICE OF THE STATE CHIEF INFORMATION OFFICER
ENTERPRISE TECHNOLOGY STRATEGIES

North Carolina Statewide Technical Architecture

Application Domain

STATEWIDE TECHNICAL ARCHITECTURE

Application Domain

Initial Released Date:	September 12, 2003	Version:	1.0.0
Revision Approved Date:			
Date of Last Review:	March 17, 2004	Version:	1.0.1
Date Retired:			
Architecture Interdependencies:			
Reviewer Notes: Reviewed and updated office title and copyright date. Added a hyperlink for the ETS email – March 17, 2004.			

Office of Enterprise Technology Strategies
PO Box 17209
Raleigh, North Carolina 27699-7209
Telephone (919) 981-5510
ets@ncmail.net

Application - Application Architecture

Principle 2.00.01 Architect applications to mimic business.

Rationale:

- The business process must be completely understood, requiring each business event be identified and the work unit servicing each event be identified.
- The organization must understand how work units cooperate to supply value when business events occur.
- The logical boundaries for applications should be drawn around units of work.
- Each unit of work is implemented with a collection of related business rules.
- Applications should respond to business events by invoking business rules.
- The state can take advantage of distributed systems capabilities to make application services available where the work takes place (i.e., in the geographic location where service must be provided).

Application - Application Architecture

Principle 2.00.02 Design applications to be highly granular and loosely coupled

Rationale:

- The designer should allow for the possibility of re-partitioning an application in the future.
- Being highly granular and loosely coupled provide flexibility in physical implementation (i.e., in the deployment of application components on different platforms).
- Highly granular, reusable application components are key to increased productivity and to rapid application deployment. The Componentware Architecture supports a highly granular design.
- The Application Communication Middleware Architecture supports a loosely coupled design.

Application - Application Architecture

Principle 2.00.03 Plan for extensibility and scalability.

Rationale:

- Most applications evolve to support new business requirements.
- Extensibility provides functional scalability.

Application - Application Architecture

Principle 2.00.04 Design application to reuse components.

Rationale:

- Applications should be built by assembling and integrating existing components, rather than by creating custom code. Shrinking cycle times do not allow for artisan programming.
- Managing component reuse is supported by the Enterprise Management Architecture. (See Enterprise Management Architecture Domain.)

Application - Application Architecture

Principle 2.00.05 Select tools based on Application Architecture.

Rationale:

- There is no such thing as one application tool that satisfies all application requirements.
- Most tools, such as user interfaces, business rules, end user reporting, on-line analytical processing, and multimedia, are oriented toward different areas of development.
- Integrated tools may blur logical application boundaries.
- Organizations need a suite of development tools to solve a variety of problems.
- Historically, project teams selected tools (e.g., Visual Basic, PowerBuilder, SQLWindows) first, and then had to live with the architecture those tools supported. That led to the problem of the tools driving the architecture (and thus the business) rather than the business requirements mandating the tools.
- Tools are just now becoming available for end-to-end design, development, and deployment of N-tier applications. As they mature, they may become the best of breed for all application tiers.

Application - Application Architecture

Principle 2.00.06 Define specific roles for programmers.

Rationale:

- Each of the following classes of programmers can specialize and use the tools best suited to their roles:
    - Business rule programmers.
    - User interface programmers.
    - Data access programmers.

Application - Application Architecture

Principle 2.00.07 N-tier Service Oriented Architecture enables rapid application deployment.

Rationale:

- Time savings, required for rapid deployment, can only occur if:
    - The application is comprised of re-usable components.
    - The application architecture is documented and well understood.
    - The infrastructure to support the application is in place.

Application - Application Architecture

Principle 2.00.08 Document the Architecture.

Rationale:

- Application architecture mimics and supports business processes, and must change when business processes change. Changes occur more frequently in business processes than in the data required to support the business.

- Project teams must focus on the process that creates the data at least as much as they focus on the data itself.
- Application Architecture is as important as Data Architecture (see the Data Domain) because data is a result of business processes.

Application - Designing and Developing Applications

Standard 2.01.01 Isolate Customizations to Purchased Software.

Rationale:

- Isolate customizations into separate modules from the purchased software itself to improve the ability to upgrade and move to new releases as required over time. For purchased line-of-business applications, loosely couple custom developed modules from the standard application software.

Application - Designing and Developing Applications

Best Practice 2.01.02 Design for the N-tier service oriented architecture.

Rationale:

- While many of the problems inherent in the state's existing monolithic and two-tier applications can be overcome by implementing applications with a three-tier architecture, large, complex projects that are anticipated to have high usage volumes and/or long life spans will be better served by an N-tier service oriented architecture.
- N-tier applications are easily modified to support changes in business rules.
- N-tier applications are highly scaleable.
- An N-tier architecture offers the best performance of any application architecture.
- Any combination of user interfaces (e.g., character, graphical, web browser, and telephone interfaces) may be implemented in an N-tier application.
- N-tier applications are less expensive to build and maintain because much of the code is pre-built and shared by other applications (see Chapter 5 - Componentware Architecture).

Application - Designing and Developing Applications

Standard 2.01.03 Develop 3-tier or N-tier Applications.

Rationale:

- All new agency applications should be developed using 3-tier or N-tier architecture in order to maximize flexibility and scalability.
- The logical separation of the tiers for: user interface(s); business rules; and data access code allows for simple, straightforward additions to each of the three tiers without undue impacts on the others.
- The logical separation of the tiers also allows for changing the platforms where the tiers are deployed, resulting in a high degree of scalability. As transaction loads, response times, or throughputs change, a tier can be moved from the platform on which it executes to another, more powerful platform - or be spread over multiple machines - without impacting the other tiers.

- While many of the problems inherent in the state's existing monolithic and two-tier applications can be overcome by implementing applications with a three-tier architecture, the goal should always be true, N-tier applications.
- Large, complex projects that have high usage volumes and/or long life spans will be better served by an N-tier service oriented architecture.
- The maximum benefits of an N-tier architecture are realized when many N-tier applications are deployed across the state, sharing common software services and offering multiple user interfaces.

Application - Designing and Developing Applications

Best Practice 2.01.01 Do not focus on platforms or deployment.

Rationale:

- Developers should not focus on where application components will execute (i.e., where they will be deployed) or what platforms they will execute on.
- System designers and operations support staff should make deployment decisions.
- Developers must avoid platform-specific or hard-coded interfaces that are difficult to change.

Application - Designing and Developing Applications

Best Practice 2.01.02 Generalize application interfaces.

Rationale:

- Generalize application interfaces.
- The code providing input and output to the user interface should be designed to provide input and output to as wide a range of interfaces as possible. This should include other applications as well as other types of user interfaces.
- Do not assume that application components will always be accessed via a graphical user interface (or any other user interface).
- Avoid assuming a specific page size, page format, layout language or user language whenever possible.

Application - Designing and Developing Applications

Standard 2.01.03 Avoid Common Gateway Interface (CGI) for business logic or to publish information to the Web.

Rationale:

- The Common Gateway Interface (CGI) does not scale, is not portable and is not easily integrated with application servers. Avoid use of CGI for information publishing, back-end applications or data access.
- Publishing information to the web with HTML or XML via java servlets reduces overhead and works in conjunction with EJB-based components.
- The use of ASP or other HTML publishing is acceptable for publishing only (not business logic) but JSP and Servlets are preferred.

Application - Designing and Developing Applications

Best Practice 2.01.04 Assign responsibility for business rules to business units.

Rationale:

- Assign responsibility for defining and maintaining the integrity of business rules to business units.
- IT staff is responsible for coding and administering the software that implements business rules in the network.
- The business units are responsible for the definition and integrity of business rules, and for communicating changes in business rules to IT.
- Every business rule should be assigned to a custodian.

Application - Designing and Developing Applications

Best Practice 2.01.05 Make business rules platform-neutral.

Rationale:

- Implement business rules in a non-proprietary, cross-platform language.
- This approach provides platform independence and portability.

Application - Designing and Developing Applications

Best Practice 2.01.06 Implement business rules as discrete components.

Rationale:

- Implement business rules as discrete executable components or services.

Application - Designing and Developing Applications
Best Practice 2.01.07 Access data through business rules.

Rationale:

- Each agency owns its own data and controls the access to it. By designing applications so business rules permit access to data, the agency will maintain better control over access.
- Data is created and used by business processes. In computer applications, data must be created, used by, and managed by the application component that automates the business process.
- Accessing data in any way other than by business processes bypasses the rules of the module that controls the data. Data is not manages consistently if multiple processes or users access it.
- Federated data should be used wherever possible to assure data accuracy and simplify data management.

Application - Designing and Developing Applications

Best Practice 2.01.08 Achieve working system first.

Rationale:

- Once the detailed application design is complete, concentrate on achieving a working system utilizing off-the-shelf components whenever possible. This will allow the system to be tested first and then optimized later.

Application - Designing and Developing Applications

Best Practice 2.01.09 Design for manageability.

Rationale:

- Design applications so they can be managed using the enterprise's system management practices and tools (see the Enterprise Management Domain).
- Applications and their components require the following management functions:
  - Software distribution.
  - Start-up, shutdown, and restart of components.
  - Starting multiple instances of a component.
  - Configuration of components.
  - Logging of component operations.
  - Communication of errors, exceptions, and unexpected events.
  - Security.
  - Installation, removal, and update of application modules.
  - Version control.

Application - Designing and Developing Applications

Best Practice 2.01.10 Adopt coding standards.

Rationale:

Adopt coding standards, in all languages, on all platforms.
Coding standards make debugging and maintenance easier. They should address (but not be limited to):

- Naming conventions for variables, constants, data types, procedures and functions.
- Code flow and indentation.
- Error and exception detection and handling.
- Source code organization, including the use of libraries and include files.
- Source code documentation and comments.
- Even the earliest code developed in a project should adhere to the standards.

Application - Designing and Developing Applications

Best Practice 2.01.11 Design for ease of testing.

Rationale:

- Design application components so they can be easily tested and debugged.
- Testing is a critical step in the development of client/server applications.
- Application components with consistent interfaces are easier to test on an application-wide basis.
- Error-handling, tracing, and checkpointing should be included.

- These functions should be implemented in the earliest phases of development.

Application - Managing Applications

Standard 2.02.01 ITS has adopted an SNMP-compliant NSM tool. All applications deployed must be designed to be managed by SNMP. (Note: any deviations must be explicitly approved by the IRMC.)

Rationale:

- By standardizing on SNMP as the instrumentation protocol, there is an opportunity for the state to benefit from reusing management instrumentation code.

Application - Managing Applications

Best Practice 2.02.01 Application management code is a candidate for re-use.

Rationale:

- Application management code is a good candidate for re-use because it should be standardized within and across applications.
- Spend time designing code to support application management requirements. This code should be reused, rather than reinvented, in every application.
- Application management is easiest if application development teams are provided code templates that include most of the basic application management functions.
- Once the statewide NSM tool is in place, the application templates should include code to enable applications to be managed by the NSM tool.

Application - Managing Applications

Best Practice 2.02.02 Design for end-to-end management.

Rationale:

- Manage the application as a whole entity by managing every component of the application and everything each component depends on. Application developers must instrument every component of the application to facilitate its management.
- Application dependencies include infrastructure (e.g., middleware, databases, and networks), other applications, and shared software components. Application teams must specify these dependencies when an application is deployed.
- Application reporting should be standards based and must be compatible with the state's NSM tool.

Application - Managing Applications

Best Practice 2.02.03 Design for proactive - rather than reactive - application management.

Rationale:

- Design for proactive -- rather than reactive -- application management.

- Proactive application management supports the business better. With proactive management, applications report potential problem conditions at predefined thresholds, before errors occur. This gives system administrators the opportunity to take corrective action to prevent an application from failing.
- While applications can be managed by administrators responding to errors, the ideal management is automatically undertaken by the NSM tools, and is proactive.
- Use thresholds to provide early alert to possible error conditions. For example, rather than sending an alarm when an application fails because its database table is full, send an alarm when the table is 90% full, so corrective action can be taken to prevent a business-impacting outage.
- Reactive application management is better than no application management at all. Reactive management is when administrators respond to errors and outages reported by applications after they have occurred.

Application - Managing Applications

Best Practice 2.02.04 Instrument applications to report the information necessary to manage them.

Rationale:

- Applications should report status, performance statistics, errors, and conditions. Decide at design time what status events the application should report to users (e.g., erroneous input); to application managers (e.g., database table 90% full); and to both (e.g., can't find needed file).
- Operations staff must be provided procedures for dealing with all conditions that are detected and reported. For example, if an application reports it can no longer access its database, operations staff must have instructions for handling the situation.
- At design time, decide the specific reporting requirements of an application module. Different applications may have different management needs, depending on their respective impact on the business the applications support.
- Applications should only report. Interpreting the reports and deciding on the appropriate response should be performed external to the application, by agents and the NSM framework.
- Application reporting should include run-time tracing to assist trouble-shooting operational problems. Tracing should be able to be turned on and off by administrators.
- If no NSM environment exists, applications should still report status to local log files that can be monitored by administrators. Applications should still be able to read and respond to commands from administrators.

Application - Managing Applications

Best Practice 2.02.05 Instrument applications to facilitate administration.

Rationale:

- Instrument applications to receive and process commands from administrators.
- Decide at design time what control operators and NSM tools should have over application components.

- Design applications to read and respond to commands from system administrators. Commands may include, but are not limited to, shut down, shutdown and restart, reconfigure yourself, and turn tracing on or off.
- Make application configurations parameter-driven, so applications can be reconfigured without recompiling and redistributing code.

Application - Accessibility

Standard 2.03.01 Utilize the latest version of the World Wide Web Consortium Web Content Accessibility Guidelines.

Rationale:

- The W3C is the international standards body for such protocols as HTML, XML, and CSS. The accessibility guidelines mirror Federal and International requirements for accessibility. The URL for the document can be found at http://www.w3.org/TR/WAI-WEBCONTENT.
- New web sites must utilize this standard 6 months after adoption.
- Existing Web sites must meet the standard 18 months after adoption.
- A note about the checkpoints contained in the standard: Each checkpoint has a priority level assigned by the W3C Working Group based on the checkpoint's impact on accessibility.

[Priority 1]

- A Web content developer must satisfy this checkpoint. Otherwise, one or more groups will find it impossible to access information in the document. Satisfying this checkpoint is a basic requirement for some groups to be able to use Web documents.

[Priority 2]

- A Web content developer should satisfy this checkpoint. Otherwise, one or more groups will find it difficult to access information in the document. Satisfying this checkpoint will remove significant barriers to accessing Web documents.

[Priority 3]

- A Web content developer may address this checkpoint. Otherwise, one or more groups will find it somewhat difficult to access information in the document. Satisfying this checkpoint will improve access to Web documents.

Some checkpoints specify a priority level that may change under certain (indicated) conditions

Application - Accessibility

Best Practice 2.03.01 Ensure graceful transformation.

Rationale:

- Pages that transform gracefully remain accessible despite any of the constraints described in the introduction, including physical, sensory, and cognitive disabilities, work constraints, and technological barriers.
- Separate structure from presentation.

- Provide text (including text equivalents). Text can be rendered in ways that are available to almost all browsing devices and accessible to almost all users.
- Create documents that work even if the user cannot see and/or hear. Provide information that serves the same purpose or function as audio or video in ways suited to alternate sensory channels as well.
- This does not mean creating a prerecorded audio version of an entire site to make it accessible to users who are blind. Users who are blind can use screen reader technology to render all text information in a page.

Application - Accessibility

Principle 2.03.01 Emerging technologies that improve accessibility should be utilized to the maximum extent possible.

Rationale:

- Technology is evolving rapidly. New access problems and solutions are appearing on a nearly daily basis.
- The rapid development of technology makes any determination of current technical feasibility extremely short lived.
- Many standards contained in this architecture are based upon human need, rather than specific solutions.
- As technology moves forward, solutions that improve accessibility are always best practices.

Application - Accessibility

Best Practice 2.03.02 Create documents that do not rely on one type of hardware.

Rationale:

- Pages should be usable by people without utilizing a mouse, with small screens, low resolution screens, black and white screens, no screens, with only voice or text output, etc.

Application - Accessibility

Principle 2.03.02 Accessibility should be proportional to the existing and potential needs.

Rationale:

This principle is intended to assure that requirements do not create unforeseen inefficiencies and create cost without significant benefit.

- Systems should be assessed for their need to be accessible to achieve the highest value.
- No standard is applied so as to go beyond what is necessary to achieve its objective. Undue Burden means significant difficulty or expense. In determining whether an action would impose an undue burden on the operation of the agency in question, factors to be considered include:
    - The nature and cost of the action needed to make a system accessible;
    - The overall size of the agency's program and resources, including the number of employees, number and type of facilities, and the size of the agency's budget;

- The type of the agency's operation, including the composition and structure of the agency's work force;
- The impact of such action upon the resources and operation of the agency.
- Statewide or Citizen services should always consider accessibility.

Application - Accessibility

Principle 2.03.03 Compatibility increases accessibility.

Rationale:

- Systems that maximize the accessibility functions of all tiers provide information in its most accessible form.
- Many people use "accessors" with limited capability to process information.
- Compatible multi-tier platforms (i.e. Microsoft's Windows Server/Professional/CE or Linux/Palm) provide opportunities to maximize accessibility.
- Software and hardware should be compatible with existing standards in Adaptive Technology.
- Government, or industry standards for hardware and software interfaces are positive, but not definitive, indicators that E&IT and accessibility technologies that are being considered for use will be compatible.

Application - Accessibility

Principle 2.03.04 Separate the Presentation Layer from the Content Layer.

Rationale:

- Accessible systems allow the client to process content in the form that best suits the client and user's capabilities.
- Users of a system are trying to access INFORMATION, not PRESENTATION.
- Systems that are utilized specifically for presentation, such as public relations material, should have their content available in an accessible format that is fundamentally as easy to obtain and use as the manner in which the content is provided inaccessibly.
- Documents should not be stored and cataloged in a manner that mandates a particular presentation (e.g. Adobe PDF), or is not in itself accessible (e.g. PCDocs).

Application - ComponentWare

Principle 2.04.01 Componentware Architecture facilitates the reuse of components across the enterprise.

Rationale:

- Reusable components increase the productivity of the application development departments within the enterprise.
- Sharing components across the enterprise greatly increases the ability of the system to meet the changing needs of the business.
- The use of proven components enhances the accuracy of information processing.

Application - ComponentWare

Principle 2.04.02 The focus of Componentware Architecture is to improve business performance.

Rationale:

- A component-based development strategy enables adaptive systems to meet the changing business needs and technical environments.
- A component-based development strategy aligns information technology with the commonly used functions of the business.

Application - ComponentWare

Principle 2.04.03 Shareable components must be callable by any application.

Rationale:

- Reusing existing shared components eliminates duplication of development, testing, and maintenance effort.
- Reusing existing shared components eliminates processing inconsistencies because business rules are maintained in one piece of code.
- Use of components reduces the time and effort required for developing and updating applications.

Application - ComponentWare

Principle 2.04.04 Components should have ownership and maintenance responsibilities assigned.

Rationale:

- The responsibility for the development and maintenance of the component remains with the application development team.
- The responsibility for the definition of a component should is within the business organization within the enterprise that is responsible for the function.

Application - ComponentWare

Principle 2.04.05 A component should implement a single business rule or function, or a small set of related business rules or functions.

Rationale:

- To maximize component reusability each should contain a single function.
- Each component must have a single point of entry and a single point of exit.

Application - ComponentWare

Principle 2.04.06 Develop reusable testing suites for every component. A component testing suite contains special programs needed for calling a component as well as sample input and example output data for verifying results.

Rationale:

- Testing suites would be developed that can be reused any time a particular component is modified.
- The testing suites will be maintained and managed the same as any other component.

Application - ComponentWare

Principle 2.04.07 New components must be platform independent.

Rationale:

- Components must be developed so they can be deployed on any supported platform.
- If the business needs change or a new platform is required, the component should easily migrate to a new platform.

Application - ComponentWare

Principle 2.04.08 Purchase rather than build components whenever possible.

Rationale:

- Purchased components must be capable of being implemented in a service-oriented environment, (i.e., can be integrated into an N-tier environment with a published Application Programming Interface (API)).
- Components should be purchased whenever possible, such as class libraries, allowing developers to focus on the development of specialized business rule components.

Application - ComponentWare

Principle 2.04.09 Design components to be fully self-contained.

Rationale:

- All necessary validation, error detection and reporting capabilities, logging/debugging/tracing functionality, monitoring and alert functionality, and systems management capabilities must be incorporated in the component.
- This facilitates operation, administration, and maintenance functions.

Application - ComponentWare

Principle 2.04.10 Establish guidelines to optimize performance.

Rationale:

- Guidelines for request and reply message lengths should be established to avoid undue network traffic and performance problems.
- Components should be compiled into a binary executable format, not interpreted.

Application - Component Reuse

Standard 2.05.01 No vendor proprietary API calls for infrastructure security services. Use Generic Security Services-API (GSS-API) or CDSA compliant API calls and products.

Rationale:

- Applications requiring security services prior to CDSA products or services being available can use the GSS-API.
- The GSS-API is an IETF standard (RFC 2078) and supports a range of security services such as authentication, integrity, and confidentiality.
- It allows for plug-ability of different security mechanisms without changing the application layer.
- It is transport independent, which means it can be used with any underlying network protocol.
- Applications using GSS-API can be retrofit to a CDSA foundation without major modifications, therefore providing an interim step to CDSA based services.

Application - Component Reuse

Best Practice 2.05.01 Establish a reuse methodology for the identification and implementation of components.

Rationale:

A methodology that supports reuse contains the following steps:

- Classify the business requirements by service type (e.g., application, interface, support, or core).
- Search the repository for reusable components that support the business or functional requirements.
- Analyze candidate components to ensure they fully meet the requirements.
- Incorporate the selected components into the new or re-engineered application using standard API's.
- Harvest new components from new or existing applications that have not been componentized yet. Placing the new component information into the repository.
- Incorporate the reuse methodology into the system development life cycle.

Application - Component Reuse

Best Practice 2.05.02 Establish a component review board to identify common components.

Rationale:

- Components used by multiple business units must be commonly understood and consistently referenced by all business users.
- Component development can be achieved through the context of projects.
- The review board should start with small, achievable, and strategic projects.
- In order to create reusable components, cooperation is needed among the business process owners.
- A framework needs to be put in place that allows for:
- Centralized management of reusable, shareable components.
- Design reviews of new and existing projects for reusable components.

- Enterprise access to information about reusable components.

Application - Component Reuse

Best Practice 2.05.03 Establish component design reviews of all ongoing projects.

Rationale:

- Determine whether the business requirements are met by any existing components.
- If the business requirements are not met by existing components, determine if there are any components that could be expanded to satisfy the business requirement (without compromising the reusability of a component).
- Analyze existing applications for potential additions to the component repository.
- Access should be provided to the proper members of application development projects to reference the component repository in order to actively research application requirements.

Application - Component Reuse

Best Practice 2.05.04 Establish a repository for maintaining the information about available reusable components.

Rationale:

- The repository provides a place to store documentation about the component API's.
- The repository should be made available to all application developers as a tool for performing their jobs.

Application - Component Reuse

Best Practice 2.05.05 Components should have ownership and maintenance responsibilities assigned.

Rationale:

- The responsibility for the development and maintenance of the component remains with the application development team.
- The responsibility for the definition of a component is within the business organization within the enterprise that is responsible for the function.

Application - Component Reuse

Best Practice 2.05.06 Every component must have a published API.

Rationale:

- A published API defines the public interface for a component or service. The API is how other applications will communicate with the component. Documentation should include input and output parameters, which parameters are required, which parameters are optional, and the lengths and types of the parameters.
- The API should be entered into the component repository that is available to all application developers.

Application - Component Reuse

Best Practice 2.05.07 Harvest components from existing applications to initially build the component repository.

Rationale:

- Legacy applications are a good resource for building a component repository.
- There is no need to reinvent a process or piece of functionality if software already exists that performs the desired function.
- If feasible, develop a wrapper that defines the API for the service and allows the legacy application to become a reusable component.

Application - Component Reuse

Best Practice 2.05.08 A component should implement a single business rule or function, or a small set of related business rules or functions.

Rationale:

- To Maximize component reusability each should contain a single function.
- Each component must have a single point of entry and a single point of exit.

Application - Component Reuse

Best Practice 2.05.09 Develop reusable testing suites for every component. A component testing suite contains special programs needed for calling a component as well as sample input and example output for verifying results.

Rationale:

- Testing suites would be developed that can be reused any time a particular component is modified.
- The testing suites will be maintained and managed the same as any other component.

Application - Component Reuse

Best Practice 2.05.10 Adopt effective component management methodologies, including the tools to support component reuse.

Rationale:

- In a distributed development environment, there must be a methodology in place for managing the available components.
- It must include the steps necessary for identifying, defining, and developing reusable components.
- If this methodology is not in place then reuse will be very difficult to implement.

Application - Component Services

Best Practice 2.06.01 Component services should be callable by any application or any other component.

Rationale:

- Components must be designed and developed with the understanding that the process that invokes it may or may not be developed in the same language or in the same environment.
- A component should be callable from any supported language on any supported platform.

Application - Component Services

Standard 2.06.01 Custom developed application components must be written in a portable, platform-independent language, such as C, C++, or Java.

Rationale:

- Application components written in a portable language are easier to move from one platform to another because they require fewer modifications to conform to the new host. This portability allows an application to be more adaptive to changes in platform technology.

Application - Component Services

Best Practice 2.06.02 New components must be platform independent.

Rationale:

- Components must be developed so they can be deployed on any supported platform.
- If the business needs change or a new platform is required, the component should easily migrate to a new platform.

Application - Component Services

Standard 2.06.02 Use statewide security infrastructure when available.

Rationale:

- Security services will be provided as a statewide infrastructure service.

Application - Component Services

Best Practice 2.06.03 Purchase rather than build components whenever possible.

Rationale:

- Purchased components must be capable of being implemented in a service-oriented environment, (i.e., can be integrated into an N-tier environment with a published Application Programming Interface (API)).
- Components should be purchased whenever possible, such as class libraries, allowing developers to focus on the development of specialized business rule components.

Application - Component Services

Standard 2.06.03 Statewide infrastructure services must be Common Data Security Architecture version 2.0 (CDSA v2.0) compliant.

Rationale:

- The CDSA version 2.0 architecture is an Open Group specification for providing security services in a layered architecture and managed by a Common Security Services Manager (CSSM). CDSA provides a management framework necessary for integrating security implementations. Version 2.0 of the specification is a cross-platform architecture, which includes a testing suite for inter-operability testing.
- A wide range of vendors has announced support for the specification and products for a broad set of platforms can be expected.
- Security protocols such as SSL, S/MIME, IPSec can all be built from a CDSA base.

Application - Component Services

Best Practice 2.06.04 Design components to be fully self-contained.

Rationale:

- All necessary validation, error detection and reporting capabilities, logging/debugging/tracing functionaility, monitoring and alert functionality, and system management capabilities must be incorporated in the component.
- This facilitates operation, administration, and maintenance functions.

Application - Component Services

Best Practice 2.06.05 Establish guidelines to optimize performance.

Rationale:

- Guidelines for request and reply message lengths should be established to avoid undue traffic and performance problems.
- Components should be compiled into a binary executable format, not interpreted.